VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Simulace BCH kódů pomocí open source nástroje Octave

Simulation of BCH Codes with Open Source Project Octave

2016 Petr Műller

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra telekomunikační techniky

Zadání bakalářské práce

Student: **Petr Műller**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2601R013 Telekomunikační technika

Téma: Simulace BCH kódů pomocí open source nástroje Octave
Simulation of BCH Codes with Open Source Project Octave

Jazyk vypracování: čeština

Zásady pro vypracování:

BCH kódy jsou využívány v komunikacích pro zabezpečení zpráv proti chybám. Cílem bakalářské práce je simulace BCH kódů v prostředí open-source nástroje Octave.

1. Nastudujte a popište základní typy samoopravných kódů.
2. Nastudujte a popište možnosti simulačního prostředí Octave.
3. Proveďte simulaci kódování BCH kódů.
4. Proveďte simulaci dekódování BCH kódů.

Seznam doporučené odborné literatury:

[1] W. Cary Huffman, Vera Pless, *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2010, ISBN 978-0521131704

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Pavel Nevlud**

Datum zadání: 01.09.2015
Datum odevzdání: 29.04.2016

doc. Ing. Miroslav Vozňák, Ph.D.
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 21. dubna 2016 .....

Rád bych poděkoval vedoucímu práce Ing. Pavlu Nevludovi za odbornou pomoc, konzultaci a čas, který si našel, aby mi pomohl k vypracování této bakalářské práce. Nadále rodině, blízkým a všem, kteří mi s prací pomohli.

Abstrakt

Bakalářská práce se zabývá simulací kódování a dekódování BCH kódů open source nástroje Octave. V teoretické části jsou popsány základní typy korekčních kódů. V dašlí části je vysvětlení principu kódování a dekódování BCH kódů. Konkrétně v dekódování je detailně zpracováno Peterson-Gorenstein-Yierlův dekódovácí algoritmus s ukázkou výpočtu a slovně zmíněny další typy dekódování.
Nadále je v práci popsána práce s prostředím GNU Octave, doinstalace balíčků potřebných k simulaci BCH kódů.
Samotná simulace BCH kódů v GNU Octave je doplněná znázorněním výstupu programu a okomentování daných kroků kódů. Simulační program byl vytvořen k zautomatizování simulace a lepšímu pochopení daných kroků, ukázce schopnosti kódu. Celkový kód je umístěn na přiloženém DVD pro volné šíření a použití v programu GNU Octave.

Klíčová slova: Galoisovo těleso, Hammingova vzdálenost, BCH kód, Peterson-Gorenstein-Yierl algoritmus, Korekční kód, Octave

Abstract

This bachelor thesis deals with simulation of coding and decoding BCH codes with open source project Octave. In the theoretical part, there are basic types of correction codes described. The analytical part explains principles of coding and decoding BCH codes. In the decoding segment, the main focus is put on the processing Peterson-Gorenstein-Yierl's decoding algorithm with a demonstration of calculation and a verbal mention of another types of decoding.
Further on, the work with open source project Octave, the installation of necessary packages for simulations BCH codes are described. The actual simulation of BCH codes in GNU Octave is completed with a presentation of a program output and a comment of its steps. The simulation program was created in order to automate the simulation, for a better understanding of the steps and a presentation of the ability of the codes. The whole code is placed on the DVD, which can be used for a free distribution and a use in GNU Octave program.

Key Words: Galois field, Hamming distance, BCH codes, Peterson-Gorenstein-Yierl algorithm, Correction code, Octave

Obsah

Seznam použitých zkratek a symbolů **8**

Seznam obrázků **9**

Seznam tabulek **10**

1 Úvod **11**

2 Základní pojmy **12**
2.1 Konečné těleso . . . 12
2.2 Galoisovo těleso $GF(p^r)$ . . . 12
2.3 Kódy . . . 13
2.3.1 Detekční kódy . . . 13
2.3.2 Korekční kódy . . . 14
2.3.3 Systematické kódy . . . 14
2.3.4 Nesystematické kódy . . . 14
2.4 Hammingova vzdálenost . . . 14
2.5 Minimální Hammingova vzdálenost . . . 14
2.6 Hammingova váha . . . 14

3 Korekční kódy **15**
3.1 Rozdělení kódů . . . 15
3.1.1 Lineární kódy . . . 15
3.1.2 Blokové kódy . . . 15
3.1.3 Cyklické kódy . . . 15
3.2 Obecně o korekčních kódech . . . 15
3.2.1 Využití korekčních kódů . . . 17
3.3 Opakovací kód . . . 17
3.4 Hammingovy kódy . . . 17
3.5 Reed-Műllerovy kódy . . . 18
3.6 Reed-Solomonovy kódy . . . 18

4 BCH kódy **20**
4.1 Kódování . . . 21
4.1.1 Příklad výpočtu kódování BCH kódů . . . 22
4.2 Dekódování . . . 23
4.2.1 Peterson-Gorenstein-Zierlerovo dekódovaní . . . 24
4.2.2 Příklad dekódování . . . 27

4.2.3 Berlekamp-Masseyův dekódovací algoritmus . . . 28
4.2.4 The Sugiyama dekódovací algoritmus . . . 29

5 Popis simulačního nástroje GNU Octave 30
5.1 Popis prostředí . . . 30
5.2 Historie . . . 30
5.3 Octave a Matlab . . . 31
5.4 Instalace . . . 31

6 Simulování kódování a dekódování v programu GNU Octave 34
6.1 Kódování . . . 34
6.2 Dekódování . . . 34
6.3 Kódování s chybou . . . 34
6.4 Dekódování s chybou . . . 35
6.4.1 BCH(15,7) $chyb \leq t$ . . . 36
6.4.2 BCH(15,7) $chyb > t$ . . . 36

7 Vytvoření simulačního programu v GNU Octave 38
7.1 Zobrazení . . . 38
7.2 Zadání zprávy . . . 38
7.3 Výběr délky . . . 39
7.4 Počet generujících se chyb . . . 39
7.5 Ověření opravné schopnosti kódu . . . 40
7.6 Výběr kódů . . . 41
7.7 Simulace s opravením zadaných $chyb > t$ . . . 41
7.7.1 Další kódová kombinace . . . 43

8 Závěr 46

Literatura 47

Přílohy 47

A CD/DVD 48

Seznam použitých zkratek a symbolů

GF	–	Galoisovo těleso
BCH	–	Bose-Ray-Chaudiri-Hocquenghem kódy
RS	–	Reed-Solomon kód
CRC	–	Cyklický redundantní součet
RM	–	Reed-Műllerův kód

Seznam obrázků

1 Komunikační kanál 16
2 Logo simulačního prostředí Octave 30

Seznam tabulek

1 Prvky Galoisova tělesa 13
2 Nerozložitelné prvky 13
3 Prvky Hammingova kódu 17
4 Prvky Reed-Műllerova kódu 18

1 Úvod

Tato práce je zaměřena na BCH kódy a jejich následnou simulaci. Cílem této bakalářské práce je simulace kódování a dekódování BCH kódů v GNU Octave a přiblížení problematiky BCH kódů. Simulace probíhá s daty pouze v binární formě.

Tato práce je rozvinuta do několika částí. Začátek práce věnuji základním pojmům týkající se přenosu dat a lineární algebry. Nadále v práci popíši pár korekčních kódů, jako Hammingův, RM nebo RS kód. Dalším popsaným tématem bude BCH kód. Popíši kódování i dekódování s uvedenými příklady výpočtu. U dekódování zaměřím pozornost na Peterson-Gorenstein-Yierl algoritmus. Další dekódovací metody budou v práci jen zmíněny.

V kapitole Octave přiblížím prostředí, historii a srovnám Octave s konkurenčním prostředím Matlab. Popsáno bude taky doinstalování balíčků, které jsou nezbytnou součástí programu při simulaci BCH kódů.

Závěr práce bude vyhrazen simulaci BCH kódů v prostředí Octave a shrnutí dříve zmíněné teorie do grafické podoby. Uvedu základní simulace k lepšímu pochopení problematiky kódů a jejich funkčnosti. V práci uvedu také výstupy z programu a popíši jednotlivé kroky programu a jejich význam. Poslední kapitolou je závěr, kde shrnu, co jsem v práci zjistil a jaké nové věci jsem se při studiu BCH kódů naučil.

2 Základní pojmy

2.1 Konečné těleso

Je algebraická stavba, která je tvořená zbytky po dělení celých kladných čísel prvočíslem p. V technické praxi hraje velkou roli zejména konečné těleso Z_2, tedy $p = 2$, které tvoří množina {0,1}.[5]

Algebraické operace součtu a násobení jsou definovány následující způsobem:

$$\begin{array}{cccc} 0+0 & 0 & 0\,.\,0 & 0 \\ 0+1 & 1 & 0\,.\,1 & 0 \\ 1+0 & 1 & 1\,.\,0 & 0 \\ 1+1 & 0 & 1\,.\,1 & 1 \end{array}$$

Operace součtu je realizována pomocí XOR. V konečném tělese Z_2 je operace a rozdíl dvou prvků stejná. Operace násobení je logický součin AND. Z konečného tělesa se následně tvoří Galoisovo těleso. [5]

2.2 Galoisovo těleso $GF(p^r)$

Úkony související se zajištěním posloupnosti nezajištěných prvků signálu se uskutečňují pomocí prvků tzv. Galoisova tělesa, které značíme $GF(p^r)$, kde nám p značí základ číselné soustavy a r je stupeň rozšíření tělesa. Galoisovo těleso je vytvářeno konečným počtem prvků $n = (p^r)$, tedy p^r odpovídá kompletnímu počtu kódových prvků v mnohočlenu zajištěné zprávy.[4]

Galoisovo těleso je těleso s konečným počtem prvků. Těleso vznikne rozšířením tzv. konečného tělesa Z_p a to stupněm r. Považuje se tím vytvoření množiny vektorů o r prvcích, kde je každý prvek je Z_p . Pro formulaci problémů v kódování se využívá Galoisovo těleso $GF(p^r)$, zejména pro binární kódy, které vznikne rozšířením tělesa Z_2. Těleso pak můžeme formulovat pomocí zbytků po dělení mnohočlenů ze Z_2. Používá se zejména v kódování dat a šifrování.[4] [6]

Pro určitý mnohočlen platí:

- je nerozložitelný v uvažovaném rozsahu okruhu mnohočlenů,
- dělí beze zbytku $(x^n - 1)$ a žádný jiný dvojčlen nižšího stupně tohoto tvaru,
- je stupně r.

K určování Galoisova tělesa GF (2^4) se berou pouze mnohočleny stupně r (v tomto případě stupně 4).
Prvky $GF(2^4)$ jsou uvedeny v tabulce 1.
Galoisovo těleso je generované $G(x) = 1 + x + x^4$

Exponenciální tvar	Polynomální tvar	Binární tvar
0	0	0 0 0 0
1	1	1 0 0 0
α	α	0 1 0 0
α^2	α^2	0 0 1 0
α^3	α^3	0 0 0 1
α^4	$\alpha + 1$	1 1 0 0
α^5	$\alpha^2 + \alpha$	0 1 1 0
α^6	$\alpha^3 + \alpha^2$	0 0 1 1
α^7	$\alpha^3 + \alpha + 1$	1 1 0 1
α^8	$1 + \alpha^2$	1 0 1 0
α^9	$\alpha^3 + \alpha$	0 1 0 1
α^{10}	$\alpha^2 + \alpha + 1$	1 1 1 0
α^{11}	$\alpha + \alpha^2 + \alpha^3$	0 1 1 1
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$	1 1 1 1
α^{13}	$\alpha^3 + \alpha^2 + 1$	1 0 1 1
α^{14}	$\alpha^3 + 1$	1 0 0 1
$\alpha^{15} = \alpha^0$	$\alpha^0 = 1$	1 0 0 0

Tabulka 1: Prvky Galoisova tělesa

Pro případ $GF(2^4)$ je přiřazení kořenů k jednotlivým nerozložitelným mnohočlenům uvedeno v tabulce níže.

i	Nerozložitelný mnohočlen	Kořeny mnohočlenu
1	$x^4 + x + 1$	$\alpha^1, \alpha^2, \alpha^4, \alpha^8$
2	$x^4 + x^3 + x^2 + x + 1$	$\alpha^3, \alpha^6, \alpha^9, \alpha^{12}$
3	$x^2 + x + 1$	α^5, α^{10}
4	$x^4 + x^3 + 1$	$\alpha^7, \alpha^{11}, \alpha^{13}, \alpha^{14}$

Tabulka 2: Nerozložitelné prvky

2.3 Kódy

Kódy rozdělujeme na dvě základní skupiny, na skupinu detekční a korekční. Ve zkratce si popíšeme každou ze skupin. Níže popsané kódy se označují jako bezpečnostní kódy.

2.3.1 Detekční kódy

Detekční kódy už podle názvu umožňují rozpoznat chybu, která vznikne během přenosu. Mohou rozpoznat jednu, ale i více chyb. Tato schopnost vzniká z nadbytečnosti dat. Typickým detekčním

kódem je CRC kód.
Detekční kódy nedokážou samy opravit ani jednu chybu, k tomu nám slouží další typ a tím jsou korekční kódy.

2.3.2 Korekční kódy

Tyto kódy můžeme občas najít taky pod pojmem FEC, tato zkratka znamená Forward Error Correction. Korekční kódy dokážou opravit jednu či více chyb. Tato vlastnost vzniká na základě vazeb mezi jednotlivými kódovými prvky. Korekční prvky potřebují zpětný kanál. Tyto kódy mají určité množství zabezpečovacích prvků.
Korekčních kódů je spousta, například Hammingův kód, označovaný jako perfektní, který dokáže opravit jenom jednu chybu vzniklou při přenosu nebo například BCH kódy, které dokážou opravit více chyb a jsou hlavní tématem této bakalářské práce. Existuje řada dalších korekčních kódů.[7]

2.3.3 Systematické kódy

Systematické kódy jsou rozdělené na část informační a část zabezpečovací přenášené zprávy. Mezi takovéto kódy patří například korekční kódy.

2.3.4 Nesystematické kódy

Nesystematické kódy na rozdíl od systematických nemají rozdělenou zvlášť informační a zabezpečovací část. Mezi takovéto kódy patří izokódy.

2.4 Hammingova vzdálenost

Hammingova vzdálenost je vzdálenost, ve které se liší libovolné dvě kódové kombinace. Je to důležitá vlastnost kódu s ohledem na jeho schopnosti, co se týče zabezpečení, protože čím větší je vzdálenost, tím je větší zabezpečovací schopnost kódu. Označuje se jako $\boldsymbol{d}$.

2.5 Minimální Hammingova vzdálenost

Udává počet míst, ve kterých se liší libovolné dvě kódové kombinace. Označuje se jako d_{min}.

2.6 Hammingova váha

Udává počet nenulových míst v kódové kombinace. Označujeme ji jako $\boldsymbol{w}$.

3 Korekční kódy

3.1 Rozdělení kódů

BCH kódy patří do skupiny cyklických, lineárních a blokových kódů, které slouží pro zabezpečení dat v telekomunikačních systémech. Popišme si tedy tyto tři skupiny blíže.

3.1.1 Lineární kódy

Umožňují realizaci efektivnějších algoritmů pro kódování a dekódování, než jiné kódy. Jsou zadány vytvářející maticí G o k řádcích a n sloupcích. Řádky matice G jsou použity libovolně lineárně nezávislé kombinace.

3.1.2 Blokové kódy

Blokové kódy mají přesně stanovené rozložení informačních a zabezepčovacích kódů míst vkódové kombinaci. Každá kombinace má délku n a určitý počet k informačních prvků. Mluvíme tedy o (n,k) kódu. Vždy platí $n > k$. Nejznámějšími jsou Hammingův kód, Reed Műllerův kód.

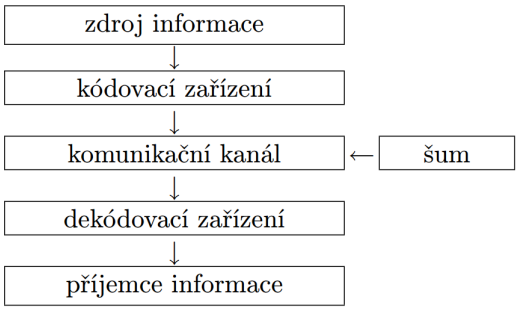
3.1.3 Cyklické kódy

Speciálním případem lineárního kódu je cyklický kód, jehož generující matice je tvořena kódovými slovy nebo-li vektory. Řádky vytvářecí matice obsahují stejné prvky, které jsou pouze posunuty o jedno místo v témže směru. Cyklické kódy se vyznačují cyklickou změnou kódového slova. Cyklické kódy jsou zadány tzv. generujícím mnohočlenem *G(x)*. Podle řádu mnohočlenu je určen počet zabezpečovacích prvků $r = (n - k)$. *G(x)* člen musí být jednoduchý. Kód má v kombinaci kódů o délce n prvků na prvních k místech nezabezpečené zprávy a na zbývajících r místech zabezpečené prvky. Kódové slovo *B(x)* odpovídá bloku informačních symbolů *M(x)* a zbytku po dělení vytvářecím mnohočlenem *G(x)*. [7]

3.2 Obecně o korekčních kódech

Korekční kódy, nebo-li samoopravné kódy, kladou důraz zejména na spolehlivý přenos dat. Při menším poškození dat nám dovolují, tato data zrekonstruovat na původní. Obsahují aplikace, jako minimalizace šumu při přehrávání disků, přenos informací po telefonních linkách, přenos informací mezi dvěma zařízeními jako například ze satelitů, nebo přenos obrázků ze sond zpět na zem. Jak velké množství těchto chyb kód opravuje, určuje jeho charakteristiku.
Prostředí, ve kterém přenášíme informace, se nazývá kanál. Jako kanál si můžeme představit například atmosféru, telefonní linky nebo elektromagnetické pole. Při průchodu dat kanálem může dojít k poškození dat, ať už třeba kvůli poruchám fyzikálního prostředí nebo jiného vlivu, který může přenášená data poškodit. Těmto poruchám obecně říkáme šum. Samotný šum může

být způsoben různými událostmi, jaké si jen lze představit. Může ho způsobovat bouřka, meteorologický roj, skvrny na slunci, přeslech telefonních linek nebo jen takovou banalitou, jako je překlep při psaní, špatnou artikulací a podobně.
Co nám může samotný šum způsobit za škody, je velice jednoduché. Může být vinen za to, že přijatá zpráva nebude shodná se zprávou odeslanou.
Samoopravné kódy se tímto problémem zabývají a snaží se odhalit a opravit chyby způsobené šumem v kanále. Můžeme si vyjádřit přenos informací na obrázku. Nejvíc nás bude zajímat část diagramu, která popisuje šum. Tento šum můžeme snížit prevencí, například zvolením vhodného komunikačního kanálu.
Šum je možno různými prevencemi zmenšit, ale nedokážeme jej zcela odstranit, proto jsou pro nás v digitálních komunikacích samoopravné kódy velice důležité. Obrázek zachycující zdroj a příjem přes komunikační kanál, kde můžeme vidět šum. [8]



Obrázek 1: Komunikační kanál

Konstrukcí kódovacího a dekódovacího zařízení sledujeme několik cílů, nejdůležitějšími jsou rychle kódování a dekódování, snadný přenos zakódované zprávy, maximum přenesených informací v kanálu za sekundu a opravu chyb způsobené šumem během přenosu zprávy. Nejpodstatnější z uvedených cílů je oprava chyb způsobené šumem během přenosu zprávy.
Každodenní forma lidského kódování je mluvená řeč. Velice často se setkáváme se šumem v běžné konverzaci, který může způsobovat sekání trávy za oknem nebo sbíječka, dělníci, kteří něco opravují a další elementy, se kterými se setkáváme v každodenním životě, aniž bychom si to uvědomovali. Tak jako samoopravné kódy, taky náš jazyk má v sobě zabudován jakousi možnost opravy tohoto šumu. Pro nás je tak běžná, že si ji ani neuvědomujeme a nedochází nám, jak často se s ní setkáváme, jak často opravujeme něčí kód nebo chcete-li, informaci přenášenou směrem k nám. Tomuto šumu dokážeme čelit díky nadbytečnosti slov, nebo hlásek některých slov. Mnoho větám lze v českém jazyce rozumět i když vynecháme většinu samohlásek, které budeme považovat za způsobenou chybu při přenosu. Proto se tipuje, že nadbytečnost jazyka je nad 50 procent. Z toho nám vyplývá fakt, že pro věci, které chceme sdělit, nám stačí méně než polovinu hlásek.
V oblasti informatiky však přenášíme informaci v binární posloupnosti. Slovo se přenáší v binár-

ním kanálu jeden po druhém. Binární kanál můžeme přirovnat ke kanálu, který jsme si uváděli u šumu, jen s tím rozdílem, že tady se přenáší binární tvar. Není jen jeden způsob přenášení těchto čísel, lze je přenášet pomocí různých pulzů (mechanických, elektrických, magnetických). Prakticky je jedno jakých, nejdůležitější je, aby bylo jednoduše rozpoznatelné, kdy je přenášena 1 a kdy 0. [8] Proč se tedy korekční kódy používají? Používáme je především, protože žádná informace není absolutně odolná vůči chybám. Někdy je toto riziko zanedbatelné, ale v mnoha případech je toto riziko vyšší.

3.2.1 Využití korekčních kódů

Škála využití samoopravných kódů je opravdu široká. Všude, kde se používá přenos z jednoho zařízení na druhé se může vyskytnout šum, nebo-li chyba při přenosu. Uveďme si pár příkladů, kde se s nimi můžeme setkat: CD přehrávače, mobilní sítě, paměti a pevné disky, čárové kódy, čipové karty, kosmický výzkum a tak dále.

3.3 Opakovací kód

Mezi nejjednodušší korekční kódy patří kód opakovací s počtem opakování n. Potřebuje největší počet zabezpečovacích prvků z korekčních kódů. Má nejmenší účinnost. Jednoduše pošle informaci několikrát po sobě, například třikrát. Pokud dojde při přenosu k chybě, tak jako správná zpráva bude vyhodnocena ta, která přijde vícekrát. Opakovací kód dokáže opravit jednu chybu, jestliže se jeho $d_{min} = 3$.[7]

3.4 Hammingovy kódy

Hammingův kód je pojmenovaný po Richardu Hammingovi a používá se v telekomunikacích pro přenos dat. Je to jeden z nejvýznamnějších kódů se schopností opravit chyby. Jeho vlastnosti jsou dány minimální hammingovou vzdáleností $d_{min} = 3$. Kód s takovouto minimální Hammingovou vzdáleností umí detekovat až dvojnásobnou chybu a opravit jednonásobnou chybu.
Hammingovy kódy jsou jednoduché na dekódování a jsou perfektní. Když o kódu řekneme, že je perfektní, znamená to, že má nejmenší myslitelnou redundanci.[10]
Když počet kontrolních bitů m zvětšujeme po jednom, celková délka Hammingových kódů bude

$$n = 2^m - 1. \tag{1}$$

Takže dostáváme poté kódy:

m	1	2	3	4
n	1	3	7	15
k	0	1	4	11

Tabulka 3: Prvky Hammingova kódu

n - počet prvků
k - počet informačních prvků
m - počet zabezpečovacích prvku

3.5 Reed-Műllerovy kódy

Reed-Műllerovy kódy jsou lineární kódy. Tyto kódy jsou pojmenovány podle svých objevitelů, byli to Irving S. Reed a DE Műller. Műller objevil tyto kódy a jeho kolega Irving S.Reed navrhl většinu dekódovací logiky. Objeveny byly v roce 1954 a v porovnání například s BCH kódy mají slabší parametry. Tyto kódy byly například sondou Mariner 9 při vysílání fotografií z Marsu. Reed-Műllerův kód oproti Hammingově kódu dokáže opravit více chyb. Minimální hammingova vzdálenost je dána:

$$n = 2^{m-r}. \tag{2}$$

Následný počet chyb, které dokáže opravit:

$$n = 2^{m-r} - 1. \tag{3}$$

Tabulka možných kombinací:

m	4	4	3	3
n	16	16	8	8
r	1	2	1	2
k	5	11	4	7

Tabulka 4: Prvky Reed-Műllerova kódu

n - počet prvků
k - počet informačních prvků
m - počet zabezpečovacích prvku

Příklad výpočtu viz. literatura [7]

3.6 Reed-Solomonovy kódy

Reed Solomon nebo-li RS kódy jsou určeny k detekci a opravě chyb. Jsou speciálním případem BCH kódů. Patří mezi blokové cyklické systematické lineární kódy. Byly objeveny v 60. letech minulého století.

Jejich největší předností je, že můžou pracovat jak s bity, tak s celými symboly. Označují se zkratkou RS *(n,k)*. Parametr k označuje počet informačních prvků, n udává velikost zpráv.[9] Počet bitových symbolů vstupujících do kodéru můžeme vyjádřit vztahem:

$$n = n - k. \tag{4}$$

Ve schopnosti dekodéru je opravit maximálně t- počet chyb. Platí vztah:

$$2t = n - k. \tag{5}$$

Minimální Hammingova vzdálenost je dána stejným vztahem, jako u BCH kódů:

$$d_{min} = 2t + 1. \tag{6}$$

Při vysílání symbolů či bitů kodér připojí ke k bitům redundantní bity. Pomocí redundantních bitů dokáže dekodér opravit vzniklé chyby ve zprávě, je-li to ovšem ve schopnosti kódu a tím získá původní data.

Kódování je obdobné jako u BCH kódů. Pro zabezpečení je nutné vytvořit generující mnohočlen *G(x)*. Po vytvoření *G(x)* dochází k samotnému procesu kódování.

Dekódování je poněkud složitější, než kódování a můžeme je rozdělit celkem do pěti fází. [1]

První zahrnuje výpočet syndromů, kde se zjistí, zda vůbec k nějaké chybě došlo. Jestli k chybě při přenosu došlo, dochází k sestavení lokátoru chyb, jehož kořeny určují na jaké pozici došlo k chybě. Jako u BCH kódů se tento krok může řešit několika algoritmy jako Euklidovým algoritmem, Berlekamp-Maseyovým algoritmem a dalšími. Po stanovení lokátoru chyb se sestaví chybový mnohočlen, stanoví se hodnoty chyby a následuje opravení chyb. [12]

Použití RS kódů je v mnoha oblastech. Stejně jako u BCH kódů je můžeme nalézt u zálohovacích mediích, přenosových protokolech digitální televize a vysoko rychlostních modemech.[9]

4 BCH kódy

Hlavním tématem této práce jsou právě BCH kódy, proto se na ně zaměříme více. Detailněji popíšeme část kódování, dekódování a uvedeme příklady výpočtu. Řekneme si také, proč se kódy nazývají BCH a kdy byly objeveny.
Celý název je Bose-Chaudriho-Hocquenghem kódy a jak můžeme vidět, název kódů vychází ze jmen autorů, proto tedy používáme zkratku BCH kódy. Je to tedy zkratka tvořena počátečními jmény autorů. BCH kódy byly objeveny v roce 1959 panem Hocequenghem a následně v roce 1960 je objevili pan Bosem a Chaudhurim nezávisle na sobě.
Značně důležitou a významnou roli nám tvoří tyto zabezpečovací korekční kódy, které jsou postavené na Hammingových kódech. Jak už jsem zmínil, korečkční kódy umožňují opravit nezávislé chyby vzniklé během přenosu. Existují dva druhy kódů, binární a nebinární. Nebinárním případem BCH kódů jsou výše zmíněné RS kódy, které jsou speciálním případem nebo-li podskupinou BCH kódů. V této práci se však budeme zabývat binárními BCH kódy, díky nimž jsou hojně využívány v digitálních zařízeních.

Mezi dobré vlastnosti těchto kódů patří velká volitelnost parametrů, tedy dobrý vztah mezi počtem informačních prvků a počtem opravených chyb. [4] [11]

Maximální délka kódového slova:

$$n = 2^m - 1, \tag{7}$$

počet informačních bitů v kódovém slově:

$$k \geq n - mt, \tag{8}$$

minimální Hammingova vzdálenost :

$$d_{min} \geq 2t + 1 \tag{9}$$

V programu Octave si můžeme díky příkazu *"bchpoly"* zobrazit volitelné parametry BCH kódu.

```
>> bchpoly
ans =

7     4     1
15    11     1
15     7     2
15     5     3
31    26     1
31    21     2
```

```
31     16      3
31     11      5
31      6      7
63     57      1
.      .      .
.      .      .
```

V tabulce můžeme vidět velkou volitelnost těchto parametrů mezi délkou slova *(n)* - první sloupec, počtem informačních prvků *(k)* - druhý sloupec a schopností kódů, kolik dokáže opravit chyb *(t)* - třetí sloupec. V bakalářské práci je ukázáno jen pár prvků z tabulky.

V programu Octave si můžeme tabulku zobrazit jednoduchým příkazem *"bchpoly"*, která nám vypíše prvky až do délky $n = 512$, která při počtu informačních prvků $k = 10$ dokáže opravit až $t = 127$ chyb.

4.1 Kódování

BCH kódy se vytvářejí metodou běžnou pro cyklické kódy. Princip zabezpečení je následující: cyklický kód, který zabezpečí zprávu, je zadán vytvářecím mnohočlenem označovaný jako *G(x)*. Galoisovo těleso $GF(2^r)$ je dáno řádem mnohočlenu, který určuje počet zabezpečovacích prvků a vypočítává se ze vztahu:

$$r = (n - k). \tag{10}$$

Po vybrání primitivního mnohočlenu a zkonstruování Galiosova tělesa, jsou následně stanoveny minimální mnohočleny $m_j(x)$ pro α^j, kde j = 1,2, ... 2t a α je symbol Galoisova tělesa. [4]

- *Z(x)* mnohočlen bloku nezabezpečené zprávy
- *G(x)* vytvářecí, nebo taky generující mnohočlen definující zabezpečovací kód
- *M(x)* mnohočlen podílu
- *R(x)* mnohočlen zbytku
- *F(x)* mnohočlen zabezpečené zprávy

Zabezpečení zprávy probíhá pomocí vztahu:

$$\frac{Z(x) * x^{n-k}}{G(x)} = M(x) + \frac{R(x)}{G(x)}, \tag{11}$$

který můžeme upravit ještě jako:

$$Z(x) * x^{n-k} = M(x) * G(x) + R(x). \tag{12}$$

Díky využití dvoustavového signálu, pro který víme, že platí pravidla algebry modulo 2, jsme schopni zabezpečený blok *F(x)* formulovat zápisem:

$$F(x) = Z(x) * x^{n-k} + R(x) = M(x) * G(x) + R(x), \tag{13}$$

$$F(x) = M(x) * G(x). \tag{14}$$

Postup při kódování je pak následující. Při kódování mnohočlenu *Z(x)* je mnohočlen nejprve vynásoben členem $x^{(n-k)}$. Poté je součin vydělen mnohočlenem *G(x)*, tím získáme zbytek *R(x)*.Tento zbytek *R(x)* přičteme ke zbytku z první provedené operace $Z(x) * x^{(n-k)}$. Následujícím způsobem vznikne zabezpečená zpráva, kterou označujeme jako *F(x)*. [4]

4.1.1 Příklad výpočtu kódování BCH kódů

Zvolíme si BCH kód(15,11), která dokáže opravit jednu chybu.
$n = 15$;
$k = 11$;
$t = 1$;

vstupní zpráva

$$Z(x) = [10010010000], \tag{15}$$

kterou taky můžeme přepsat jako:

$$Z(x) = (1 + x^3 + x^6), \tag{16}$$

vstupní zprávu rozšíříme pomocí x^r, přičemž r vypočítáme jako *(n-k)*. V našem případě je r = (15-11), $r = 4$.
Takže dostáváme

$$Z(x) * x^r = (1 + x^3 + x^6) * x^4 = x^{10} + x^7 + x^4, \tag{17}$$

tuto zprávu vydělíme generujícím polynomem *G(x)*, který je

$$G(x) = (1 + x + x^4). \tag{18}$$

Vydělíme vstupní zprávu *Z(x)* generujícím polynomem *G(x)*

$$\frac{x^{10} + x^7 + x^4}{1 + x + x^4}, \tag{19}$$

z kterého dostaneme po vydělení zbytek

$$x^3 + x^2 + x + 1, \tag{20}$$

a přičtením tohoto zbytku k rozšíření $Z(x) * x^r$ dostáváme:

$$x^{10} + x^7 + x^4 + x^3 + x^2 + x + 1, \tag{21}$$

přičemž po převedení na binární tvar dostaneme *F(x)*, nebo-li vyslané slovo:

$$F(x) = [111110010010000]. \tag{22}$$

Vidíme, že naše zpráva se nachází na pozici 5. až 15. a redundantní bity jsou před nimi. Pro kontrolu můžeme použít simulaci z programu Octave:

```
>> n = 15;
>> k = 11;
>> t = 1;
>> msg = [1 0 0 1 0 0 1 0 0 0 0];
>> code = bchenco (msg, n ,k)
code =

1   1   1   1   1   0   0   1   0   0   1   0   0   0   0
```

Po simulaci vidíme, že náš postup výpočtu byl správný.

4.2 Dekódování

V této kapitole si představíme tři algoritmy, které jsou nejpoužívanější k dekódování BCH kódů. První metoda je známá jako Peterson-Gorenstein-Zierlerovo dekódování. Původně byly vytvořeny pro binární BCH kódy Petersonem v roce 1960 a zobecněny krátce potom Gorensteinem a Zirlerem. Popíšeme si tuto metodu, která obsahuje čtyři kroky. Druhý krok této procedury je nejkomplikovanější a nejvíce časově náročný.
Druhá metoda je známá jako Berlekam-Masseyovo dekódování. Tato metoda představuje efektivnější přístup ke kroku dva, který je u předchozího algoritmu složitější. Tato dekódovací metoda byla vynalezena panem Berlekampem v roce 1967. Třetí dekódovací algoritmus objevili panové Sugiyama, Kasahara, Hirasawa a Namekawa v roce 1975, je to také alternativní metodou k vykonání druhé metody Peterson-Gorenstein-Zierlerova algoritmu známého taky jako Sugiyamův algoritmus. V této sekci si také představíme hlavní myšlenky dekódovacích algoritmů, které můžou být implementovány také na Reed-Solomonovy kódy. [1]

4.2.1 Peterson-Gorenstein-Zierlerovo dekódovaní

Nechť je C BCH kód nad $F_{(g)}$ BCH kód má délku n a navrženou vzdálenost δ. Kód dokáže opravit $t = [\frac{\delta-1}{2}]$. Peterson-Gorenstein-Zierlerův dekódovací algoritmus dokáže opravit t chyb. Algoritmus obsahuje čtyři kroky, které nejprve popíšeme a pak shrneme. Tato dekódovací metoda je založena na inverzi matice syndromové.
Předpokládáme, že *y(x)* je přijaté slovo, které předpokládáme, že se liší od kódového slova vyslaného *c(x)* v nejvýše *t*souřadnicích. Proto *y(x)= c(x)+ e(x)*, kde $c(x) \in C$ a *e(x)* je chybový vektor. Předpokládejme, že chyby se vyskytují v neznámých souřadnicích $k_1, k_2, k_3 \ldots, k_v$. Potom pak [1]

$$e(x) = e_{k1}x^{k1} + e_{k2}x^{k2} + \ldots + e_{kv}x^{kv}. \tag{23}$$

Jakmile určíme *e(x)*, která nám pozici chyby k_j a veličinu e_{kj}, můžeme dekódovat přijatý vektor jako

$$c(x) = y(x) - e(x). \tag{24}$$

Definujeme si syndromy S_i z *y(x)*, jako element $S_i = y(\alpha^i)$
Prvním krokem v algoritmu je vypočítání syndromů $S_i = y(\alpha^i)$ pro $1 \leq i \leq 2t$ z přijatého vektoru. S_i definujeme jako $y(\alpha^i)$.

Nechť matice bude t x n

$$H = \begin{pmatrix} 1 & \alpha & \alpha^2 & \alpha^{n-1} \\ 1 & \alpha^2 & \alpha^4 & \alpha^{(n-1)2} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \alpha^t & \alpha^{2t} & \alpha^{(n-1)t} \end{pmatrix} \tag{25}$$

Matice je $Hy^t = S^t$, tedy nulová.

Syndromy, které nám urči systém rovnic obsahující chybu na neznámém místě a neznámé veličiny určíme tímto vztahem:

$$S_i = y(\alpha^i) = \sum_{j=1}^{v} e_{kj}(\alpha^i)^{kj} = \sum_{j=1}^{v} e_{kj}(\alpha^{kj})^i, \tag{26}$$

pro $1 \leq i \leq 2t$. Pro zjednodušení zápisu, pro $1 \leq j \leq v$, nechť je $E_j = e_k$ označující chybu veličiny k_j a $X_j = \alpha^{kj}$ označují polohu chyby. Po zjednodušení se ze zápisu vznikne vztah:

$$S_i = \sum_{j=1}^{v} E_j(X_j^i), pro 1 \leq i \leq 2t. \tag{27}$$

Předešlý vztah nás vede k systému rovnic:

$$
\begin{aligned}
S_1 &= E_1X_1 + E_2X_2 + \ldots + E_vX_v \\
S_2 &= E_1X_1^2 + E_2X_2^2 + \ldots + E_vX_v^2 \\
S_3 &= E_1X_1^3 + E_2X_2^3 + \ldots + E_vX_v^3 \\
&\vdots \\
S_{2t} &= E_1X_1^{2t} + E_2X_2^{2t} + \ldots + E_vX_v^{2t}.
\end{aligned}
\tag{28}
$$

Tento systém je nelineární v X_j s neznámým koeficientem E_j. Proto použijeme nové proměnné $\sigma 1, \sigma 2, \ldots, \sigma v$ které nám sestaví lineární systém, která vede přímo ke vztahu chybového lokalizačního mnohočlenu:

$$
\sigma(x) = (1 - xX_1)(1 - xX_2)\ldots(1 - xX_v) = 1 + \sum_{j=1}^{v} \sigma_i x^i. \tag{29}
$$

Jak bude znám chybový lokalizační mnohočlen, vracíme se k rovnici 28.

Kořeny $\sigma(x)$ jsou inverzní k umístění chyb, pak tedy:

$$
\sigma(X_j^{-1}) = 1 + \sigma_1 X_j^{-1} + \sigma_2 X_j^{-2} + \ldots \sigma_v X_j^{-v} = 0, \tag{30}
$$

pro $1 \leq j \leq v$. Vynásobením $E_j X_j^{i+v}$ vytvoříme:

$$
E_j X_j^{i+v} + \sigma_1 E_j X_j^{i+v-1} + \ldots + \sigma E_j X_j^i = 0, \tag{31}
$$

pro všechny i. Sčítáním nad $1 \leq j \leq v$ dostáváme:

$$
\sum_{j=1}^{v} E_j X_j^{i+v} + \sigma_1 \sum_{j=1}^{v} E_j X_j^{i+v-1} + \ldots + \sigma_v \sum_{j=1}^{v} E_j X_j^i = 0, \tag{32}
$$

tak dlouho dokud $1 \leq i$ a $i+v \leq 2t$, tyto součty jsou syndromy obsažené v 27, protože $v \leq t$. Z rovnice 32 se stane

$$
S_{i+v} + \sigma_1 S_{i+v-1} + \sigma_2 S_{i+v-2} + \ldots + \sigma_v S_i = 0, \tag{33}
$$

nebo

$$
\sigma_1 S_{i+v-1} + \sigma_2 S_{i+v-2} + \ldots \sigma_v S_i = -S_{i+v}, \tag{34}
$$

platí-li tedy pro $1 \leq j \leq v$, pak můžeme najít σ_k, pokud budeme řešit maticové rovnice

$$\begin{bmatrix} S_1 & S_2 & S_3 & \dots & S_{v-1} & S_v \\ S_2 & S_3 & S_4 & \dots & S_v & S_{v+1} \\ S_3 & S_4 & S_5 & \dots & S_{v+1} & S_{v+2} \\ & & & \vdots & & \\ S_v & S_{v+1} & S_{v+2} & \dots & S_{2v-2} & S_{2v-1} \end{bmatrix} \begin{bmatrix} \sigma_v \\ \sigma_{v-1} \\ \sigma_{v-2} \\ \vdots \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_{v+1} \\ -S_{v+2} \\ -S_{v+3} \\ \vdots \\ -S_{2v} \end{bmatrix} \tag{35}$$

která vzniká z rovnice 35. Naším druhým krokem v algoritmu je vyřešit rovnici 35 pro $\sigma 1, \sigma 2, \dots, \sigma v$. Jakmile je tento druhý krok u konce, je stanovena $\sigma(x)$. Nicméně, stanovení $\sigma(x)$ je komplikované, protože nevíme v. Hledáme řešení, které obsahuje nejmenší hodnotu v a tento následující krok podporuje lemma. [1]

Nechť je dán $\mu \leq t$, pak je

$$M_\mu = \begin{pmatrix} S_1 & S_2 & \dots & S_\mu \\ S_2 & S_3 & \dots & S_{\mu+1} \\ \vdots & \vdots & \vdots & \vdots \\ S_\mu & S_{\mu+1} & \dots & S_{2\mu-1} \end{pmatrix} \tag{36}$$

Pak je M_μ nesingulární jestli $\mu = t$ a singulární jestli $\mu > t$, kde v je počet chyb, které se vyskytly.
K vykonání našeho druhého dekódovacího kroku se snažíme odhadnout hodnotu chyb v. Předpokládáme, že k vykonání našeho druhého dekódovacího kroku se snažíme odhadnout hodnotu chyb v. Předpokládáme, že $\mu = t$, což je největší, které v může být. Koeficient matice 35 se snažíme řešit $M_\mu = M_t$ v 36. Jestli je M_μ singulární, snížíme naši hodnotu $\mu = t - 1$ a rozhodneme, zda naše $M_\mu = M_t$ není singulární. Děláme to tak dlouho, dokud nezískáme jedinečnou matici. Pokračujeme ve snižování našeho μ dokud nedostaneme nějaké M_μ, které bude nesingulární. S $v = \mu$ řešíme rovnici 35 a tím určíme $\sigma(x)$.

Třetím krokem je pak najít kořeny $\sigma(x)$ a invertovat je pro převedení na chybový lokalizační mnohočlen. To se většinou provádí vyčerpávající metodou kontrolování $\sigma(\alpha^i)$ pro $1 \leq i \leq n$.[1]

Čtvrtým krokem je vložit naše čísla do rovnice 28 a vyřešit v tomto lineárním systému určenou chybovou hodnotu E_j. Ve skutečnosti pouze zvážíme první rovnici v 28 kvůli následujícího důvodu. Koeficient matice z první v rovnice má determinant

$$det \begin{pmatrix} X_1 & X_2 & \dots & X_v \\ X_1^2 & X_2^2 & \dots & X_v^2 \\ \vdots & \vdots & \vdots & \vdots \\ X_1^v & X_2^v & \dots & X_v^v \end{pmatrix} = X_1 X_2 \dots X_v det \begin{pmatrix} 1 & 1 & \dots & 1 \\ X_1 & X_2 & \dots & X_v \\ \vdots & \vdots & \vdots & \vdots \\ X_1^{v-1} & X_2^{v-1} & \dots & X_v^{v-1} \end{pmatrix} \tag{37}$$

Pravá strana je determinant z determinantu Vadermondovy matice.
Podle [1] je Peterson-Gorenstein-Zierlerovo postup pro dekódovaní je proto následující:

1. Vypočítat syndromy $S_i = y(\alpha^i)$ pro $1 \leq i \leq 2t$.
2. Rozhodneme zda M_μ je singulární, zastavíme se tam, kde hodnota μ v M_μ nesingulární. Pro $\mu = v$ řešené ve vztahu 35.
3. Najít kořeny z $\sigma(x)$ spočítané $\sigma(\alpha^i)$ pro $0 \leq i \leq n$. Invertovat kořeny k získání polohy chyb X_j.
4. Řešit první rovnici 28 k získání chybových hodnot X_j.

Předpokládáme, že zakódovaná zpráva, která se vyšle se bude lišit od zprávy přijaté $v \leq t$ souřadnicích. Proto existuje jen jedna správná sada chybových umístění čísel a jedna správná sada chybových veličin, které vedou k jedinečnému polynomu, který určuje lokátor chyb.

Krok druhý musí určit správnou hodnotu v rovnicí 36, v je největší hodnota, která je menší nebo rovna t tak, že M_v je nesingulární.
Jakmile víme počet chyb, tak je řešíme za pomoci vztahu 35, abychom získali optimální řešení pro neznáme koeficienty lokátoru chyb. Krok 2. nám tedy správně určí chybový polynom a tudíž krok 3. správně určí čísla, která mají být na místě, kde se vyskytla chyba. Jakmile jsou vypočítány, první rovnice v z rovnice 28 má unikátní řešení pro chybové hodnoty, které krok 4. počítá.[1]

V případě, že obdržíme kódové slovo, které je větší než t, dekodér vyhodnotí, že došlo k více chybám a zpráva bude dekódována špatně.

4.2.2 Příklad dekódování

Uvedeme si příklad dekódování pro binární BCH kód (15,7). Navrhovanou vzdálenost budeme mít $\delta = 5$, který má definované pro T = {1, 2, 3, 4, 6, 8, 9, 12}. Použitím primitivních prvků α z tabulky 1. dostaneme generující polynom.

$$g(x) = 1 + x^4 + x^6 + x^7 + x^8.$$

Předpokládejme, že přijaté kódové slovo je

$$y(x) = 1 + x + x^5 + x^6 + x^9 + x^{10}.$$

Použitím tabulky 1. a teorému 28 dostaneme první deódovácí krok.

$$\begin{aligned} S_1 &= 1 + \alpha + \alpha^5 + \alpha^6 + \alpha^9 + \alpha^{10} = \alpha^2, \\ S_2 &= S_1^2 = \alpha^4, \\ S_3 &= 1 + \alpha^3 + \alpha^{15} + \alpha^{18} + \alpha^{27} + \alpha^{30} = \alpha^{11}, \\ S_4 &= S_2^2 = \alpha^8. \end{aligned}$$

Krokem 2. dostaneme

$$M_2 = \begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} = \begin{bmatrix} \alpha^2 & \alpha^4 \\ \alpha^4 & \alpha^{11} \end{bmatrix},$$

je nesingulární s inverzí

$$M_2^{-1} = \begin{bmatrix} \alpha^8 & \alpha \\ \alpha & \alpha^{14} \end{bmatrix}.$$

Tedy vznikly v = 2 chyb, a musíme vyřešit

$$\begin{bmatrix} S_1 & S_2 \\ S_2 & S_3 \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} -S_3 \\ -S_4 \end{bmatrix},$$

nebo

$$\begin{bmatrix} \alpha^2 & \alpha^4 \\ \alpha^4 & \alpha^{11} \end{bmatrix} \begin{bmatrix} \sigma_2 \\ \sigma_1 \end{bmatrix} = \begin{bmatrix} \alpha^{11} \\ \alpha^8 \end{bmatrix}.$$

Řešením je $[\sigma_2\sigma_1]^T = M_2^{-1}[\alpha^{11}\alpha^8]^T = [\alpha^{14}\alpha^2]^T$. Tedy krok 2. nám vytvoří chybový lokalizační polynom $\sigma(x) = 1 + \alpha^2 x + \alpha^{14}x^2$. Potom krok 3. vytváří kořeny α^{11} a α^5 ze σ a proto chyby vznikly v $X_1 = \alpha^4$ a $X_2 = alpha^{10}$. Protože se jedná o binární BCH kódy, přeskakujeme krok 4. Přičtením chybového vektoru $e(x) = x^4 + x^{10}$ k přijaté zprávě $y(x) = 1 + x + x^5 + x^6 + x^9 + x^{10}$ dostáváme původní odeslanou zprávu. Odeslané kódové slovo je $c(x) = 1 + x + x^4 + x^5 + x^6 + x^9$. Další výpočty jsou uvedeny v literature [1].

4.2.3 Berlekamp-Masseyův dekódovací algoritmus

Tento dekódovací algoritmus se liší od dříve zmiňovaného Peterson-Gorenstren-Zierlerova algoritmu ve druhém dekódovacím kroku. Ověření, jak vlastně druhý krok probíhá, je poměrně technický náročný a je uvedený v literatuře viz.[13]. V případě aplikování algoritmu na BCH kódy je jednodušší, když budou v binární podobě. Dekódování vyvinul pan Berlekamp a krátce poté pan Massey ukázal, že tento dekódovací algoritmus skutečně poskytuje nejkratší možný opakovací vztah, který generuje $S_1, S_2, \ldots$

Zaujmeme stejný způsob zápisu jako v předchozím Peterson-Gorenstren-Zierlerova algoritmu v kroku 2. V tomto kroku se poloha chyby vypočítá tím, že řeší systém v lineárních rovnic ve v neznámých, kde v je číslo chyby, která vznikla. Pokud je v velké číslo, tak je tento krok časově náročný. Pro binární kódy tento algoritmus sestavuje chybový polynom tím, že vyžaduje, aby jeho koeficienty vyhovovaly soustavě rovnic nazývané Newtonova identita, radši než tomu bylo v 34. Ukázka řešení je v literatuře [1].

4.2.4 The Sugiyama dekódovací algoritmus

Sugiyamův dekódovací algoritmus je další metodou k nalezení chybového polynomu a tudíž představuje další alternativu druhého kroku Peterson-Gorenstein-Zierlerova algoritmu. Algoritmus se vztahuje na třídu kódů nazývané Goppa kódy patřící do podtřídy BCH kódů. Dekódování je relativně jednoduché, ale taky rychlé, díky aplikování Euklidova algoritmu. Příklad dekódovacího výpočtu je opět uveden v [1]

5 Popis simulačního nástroje GNU Octave

5.1 Popis prostředí

Simulační prostředí Octave je označován jako jazyk vysoké úrovně. Poskytuje pohodlné rozhraní příkazového řádku pro řešení lineárních a nelineárních problémů a provádění jejich numerických experimentů pomocí jazyka, který je většinou slučitelný s programem Matlab.
Octave má rozsáhlé nástroje pro řešení běžných problémů numerické lineární algebry, hledání kořenů nelineárních rovnic, integrace běžné funkce, manipulace s polynomy a integrace diferenciálních rovnic. Je snadno rozšiřitelný a lze snadno přizpůsobit pomocí uživatelsky definovaných funkcí napsaných v Octave. Nebo pomocí dynamicky načtené moduly psané v C++, C, Fortran, nebo jiných jazyků. [2]

Obrázek 2: Logo simulačního prostředí Octave

5.2 Historie

Jméno programu je odvozeno od jména jednoho z bývalých autorových profesorů, který napsal slavnou učebnici o chemických reakcích a který byl také dobře znám. Jmenoval se Octave Levenspiel.
Program Octave byl původně koncipován okolo roku 1988, jako učebnice pro vysokoškoláky na navrhování chemického reaktoru. Projekt na vytvoření simulačního prostředí zahájil James B. Rawlings z University of Wisconsin-Madison, ale největší podíl na jeho vyvinutí měl John G. Ekerdt z University of Texas, který vložil hodně úsilí do tohoto projektu. Nejdříve chtěli vybudovat několik velmi specializovaných nástrojů pro řešení problémů konstrukcí chemického reaktoru. Později ale viděli omezení, které jim programy přináší. Rozhodli se proto k vybudování mnohem komplexního a použitelnějšího nástroje, kterým se Octave bezpochyby stal.[2]
Existovala skupina lidí, která říkala, že by místo toho měli používat programovací jazyk Fortran, protože to je programovací jazyk, který se používá ve strojírenství, ale pokaždé, když se studenti snažili sestavit kód, tak strávili příliš mnoho času nad tím, proč jejich Fortran kód nejede a nezbylo tedy dost času na to, aby se učili o chemickém inženýrství. Proto vznikl program Octave. Vývojáři věřili, že s interaktivním prostředím, jako je v Octave, by většina studentů měla rychle zvládnout základy a do pár hodin učení by ho měla být schopna užívat.[14]
Úplný vývoj pak začal až na jaře v roce 1992. První Alpha vydání bylo uvedeno už 4. ledna v roce 1993 a verze 1.0 byla vydána 17. února 1994. Samozřejmě od té doby program Octave

prošel několika významnými revizemi, jako je součástí Debian GNU/Linux a další.
Nyní je zřejmé, že program Octave je mnohem více, než jen výukovým balíčkem s omezenou užitečností mimo učebnu.
Jak jsme si uváděli, prostředí Octave je multifunkční nástroj pro sofistikovanou numerickou analýzu. Nabídne nám velké množství souborů s vestavěnými funkcemi, které dokážou řešit mnoho rozdílných problémů. Jedna z velkých výhod prostředí je, že umí vykreslovat různá zařízení.
Octave je oficiální projekt, který má zdrojový kód uvedený pod obecnou veřejnou licencí. Jednoduše řečeno, znamená to, že můžete software volně používat za jakýmkoli účelem, ať už kopírování nebo volnému šíření. S tímto programem, který si lze stáhnout na oficiální stránce GNU Octave. Těmto softwarům říkáme Open Source, neboli přeloženo do češtiny, volně šířící programy, které může kdokoliv používat, šířit, nahlížet do zdrojového kódů a například ho zlepšovat. Můžete potom nový software dát na stránky pod veřejnou licencí. Jestli používáte prostředí Octave jako programovací jazyk, můžete si svoji verzi rozšířit o další funkce.[2] [14]

5.3 Octave a Matlab

Určitě stojí za to zmínit program Matlab a porovnat s programem Octave. To, že jsou si oba programy velice blízko a podobné je na první pohled vidět. Také proto je Octave často vedený jako tzv. Matlab klon. I když toto přirovnání nejspíš není nejlepší a vymysleli ho vývojáři MathWorks. Octave se snaží být kompatibilní s programem Matlab. Ačkoliv jsou si vědomi, že v mnoha případech to nebude nejjednodušší.

5.4 Instalace

Program Octave si uživatel musí přizpůsobit podle toho, k čemu ho bude používat. S tím souvisí i následné stahování balíčku. Tyto balíčky člověk stáhne z oficiální stránek GNU Octave a může si o nich přečíst k čemu všemu je může použít. Jestli se tak chystáte, určitě skončíte na stránkách Octave Forge, kde lze všechny balíčky najít i informacemi k nimi určených.
Simulaci jsme tedy prováděli v programu Octave, který lze stáhnout z oficálních stránek GNU Octave pro operační systém Linux, OS X a Windows. Pro simulaci jsem používal operační systém Windows.

Po úspěšné instalaci programu jsem musel stáhnout balíčky s kterými budu pracovat a které budeme potřebovat pro simulaci našich kódů. Našli jsme, že námi požadované funkce *"bchenco"* pro zakódování zprávy a *"bchdeco"* pro dekódování zprávy obsahuje balíček *Communications*. Do *Command Windows* jsme zadali příkaz *"pkg install -forge package name"* v našem případě *pkg install -forge communications*. Po zadání příkazu ale vidíme, že balíček *Communications* potřebuje pro svou instalaci mít nainstalovaný další balíček, a to balíček *Signal*. Pro úspěšnou instalaci našeho balíčku musíme stáhnout balíček *Signal* a poté je stejným příkazem *pkg install*

-forge Signal nainstalovat. Opět vidíme, že se balíček odkazuje na další z balíčku, a to na balíček *Control*, který najdeme opět na oficiálních stránkách Octave. Každý balíček, který chceme nainstalovat, si také říká o verzi následujícího balíčku, kterou potřebuje. Viz. obrázek 188 *"Signal needs control >= 2. 4. 5"*.
Touhle metodou doinstalovávání balíčku pokračujeme, až dojdeme k balíčku, na který se předchozí balíček odkazuje. Následně vzestupnou formou postupujeme až k balíčku *Communications*. Pro naši simulaci jsme tedy potřebovali 3 balíčky, které jsme stáhli a následně nainstalovali. [15]

```
>> pkg install -forge communications
error: the following dependencies were unsatisfied:
communications needs signal >= 1.1.3
>> pkg install -forge signal
error: the following dependencies were unsatisfied:
signal needs control >= 2.4.5
>> pkg install -forge control
For information about changes from previous versions of the control
package, run 'news control'.
>> pkg list
Package Name  | Version | Installation directory
--------------+---------+----------------------
control  |   3.0.0 | C:\Octave\Octave-4.0.0\share\octave\packages
\control-3.0.0
>> pkg install -forge signal
For information about changes from previous versions of the signal
package, run 'news signal'.
>> pkg list
Package Name  | Version | Installation directory
--------------+---------+----------------------
control  |   3.0.0 | C:\Octave\Octave-4.0.0\share\octave\packages
\control-3.0.0
signal  |   1.3.2 | C:\Octave\Octave-4.0.0\share\octave\packages
\signal-1.3.2
>> pkg install -forge communications
warning: doc_cache_create: unusable help text found in file
'commsimages'
For information about changes from previous versions of the
communications package, run 'news communications'.
>> pkg list
Package Name    | Version | Installation directory
```

```
---------------+---------+----------------------
communications  |   1.2.1 | C:\Octave\Octave-4.0.0\share\octave
\packages\communications-1.2.1
control  |   3.0.0 | C:\Octave\Octave-4.0.0\share\octave\packages
\control-3.0.0
signal  |   1.3.2 | C:\Octave\Octave-4.0.0\share\octave\packages
\signal-1.3.2
>> pkg load control
>> pkg load signal
```

Abychom jsme balíček communications nemuseli nahrávat pokaždé, když zapne program Octave, tak použijeme tento příkaz

```
>>  pkg rebuild -auto communications
```

který nám zjednoduší práci s programem tak, že nám balíček při zapnutí Octave vždycky načte.
Po úspěšné instalaci můžeme vyzkoušet, jestli námi příslušný balíček, který jsme stáhli, využívá požadovanou funkci. Vyzkoušení je jednoduché. Stačí zadat požadovanou funkci, a to třeba *"bche"* nebo *"bchd"* a následný zmačknutím tabulátoru. Jestli se příkaz doplní, program využívá funkci. Instalace jen určitého množství balíčku je jednou z výhod, protože máme přehled o tom, co do programu nahráváme, a také co využijeme pro simulování.

6 Simulování kódování a dekódování v programu GNU Octave

6.1 Kódování

Simulaci začneme nejjednodušším typem **BCH(7,4)**

```
>> n = 7;
>> k = 4;
>> t = 1;
>> msg = [0 0 0 1];
>> code = bchenco (msg, n, k)
code =
1   0   1   0   0   0   1
```

Zde můžeme vidět, jak se nám vysílané slovo *"msg = [0 0 0 1]"* zakódovalo pomocí BCH kódů. Postup byl takový, že jsme si vytvořili vysílanou zprávu, která musela obsahovat 4 znaky. Následně jsme použili funkci *"bchenco"* , kde jsme museli zadat naši zprávu, délku slova (n) a počet informačních prvků (k). Na pozicích 4-7 je naše zpráva.

6.2 Dekódování

```
>> [msg, error] = bchdeco (code, k, t)
msg =
0   0   0   1


error = 0
```

Při dekódovaní jsme použili fukci *"bchdeco"*, kde jsme museli použít zakódované slovo *(code)*, počet informačních prvků(k) a jednička nám symbolizuje kolik chyb je schopný opravit. Vidíme, že dekódována zpráva se shoduje s odeslanou. Pro kontrolu jsme si nechali vypsat *"error"*, který nám ukazuje počet opravených chyb.

6.3 Kódování s chybou

Opět si pro jednoduchost zvolíme **BCH kód(7,4)** a ukážeme si, jestli dokáže opravit vzniklou chybu při přenosu.

```
>> n = 7;
>> k = 4;
>> t = 1;
>> msg = [0 0 0 1];
>> code = bchenco (msg, n, k)
```

```
code =
1   0   1   0   0   0   1

>> code_error = [1 0 1 1 0 0 1]
code_error =
1   0   1   1   0   0   1
```

Znovu jsme si uvedli vysílací slovo *msg = [0 0 0 1]* a zakódovali jsme ho. Pro chybový vektor jsme použili *code error*, který obsahuje chybu vyskytující se na pozici 4.

6.4 Dekódování s chybou

```
>> [msg, error] = bchdeco (code_error, k, t)
msg =
0   0   0   1

error =  1
```

První, co jsme udělali bylo, že jsme změnili zakódované slovo tak, aby se v něm nacházela chyba. Důvodem je ověření, zda tento kód dokáže opravit vzniklou chybu. Při dekódování kódu s chybou, se nám díky vlastnostem BCH(7,4) opravila chyba vzniklá na 4. pozici. Vidíme, že výsledný dekódovaný kód se shoduje s kódem odeslaným a *"error"* nám vypsal jednu vzniklou chybu.
Co když ale počet chyb v kódů překročí schopnost opravy kódu. Pojďme si proto na následujícím příkladu ukázat takovýto případ, kdy chyb při přenosu bude více.

```
>> n = 7;
>> k = 4;
>> t = 1;
>> msg = [0 0 0 1];
>> code = bchenco (msg, n, k)
code =
1   0   1   0   0   0   1

>> code_error = [1 1 1 0 0 0 0]
code_error =
1   1   1   0   0   0   0

>> [msg, error] = bchdeco (code_error, k, t)
msg =
0   0   1   0
```

```
error =  1
```

Nasimulovali jsme si chybu jako *"code error"* na pozici 2. a 7. Ověřili jsme si, že kód nedokáže opravit více chyb, protože se vstupní slovo nerovná dekódovanému. Při přenosu vzniklo více chyb, než jedna a to bylo nad schopnosti kódu. Dekódoval nám výsledné slovo špatně. I když nám *"error"* vypisuje, že vznikla chyba, nedokáže už při větším množstvím chyb detekovat kolik chyb nastalo a dekóduje nám výsledné slovo špatně.

6.4.1 BCH(15,7) *chyb* $\leq t$

Kódování s délkou $n = 15$ a počtem informačních prvků $k = 7$ nám dovoluje opravit 2 nezávislé chyby.

```
>> n = 15;
>> k = 7;
>> t = 2;
>> msg =[0 0 1 1 0 1 1];
>> code = bchenco (msg, n, k)
code =
1   1   1   0   0   1   1   0   0   0   1   1   0   1   1

>> code_error = [1 1 1 0 0 1 1 0 0 1 0 1 0 1 1]
code_error =
1   1   1   0   0   1   1   0   0   1   0   1   0   1   1

>> [msg, error] = bchdeco (code_error, k, t)
msg =
0   0   1   1   0   1   1

error =  2
```

V prvních krocích jsme si zadali délku, počet informačních prvků a počet chyb, které dokáže opravit. Zvolili si vysílanou zprávu a zakódovali ji. Opět jsme pomocí *"code error"* vytvořili dvě chyby, abychom vyzkoušeli schopnost kódu. Jak můžeme vidět, dekódována zpráva se rovná přijaté, tedy BCH kód (15,7) dokáže opravit 2 chyby, které nám následně vypsal *"error"*, a vypsal nám počet chyb, které opravil. Tímto jsme ověřili schopnost kódu s opravou dvou chyb.

6.4.2 BCH(15,7) *chyb* $> t$

Vyzkoušejme si, co se stane, vyskytne-li se ve zprávě víc chyb, než dokáže kód(15,7) opravit.

```
>> n = 15;
>> k = 7;
>> t = 2;
>> msg =[0 0 0 0 0 0 1];
>> code = bchenco (msg, n, k)
code =
0   0   0   1   0   1   1   1   0   0   0   0   0   0   1

>> code_error = [1 1 1 1 0 1 1 1 0 0 0 0 0 0 1]
code_error =
1   1   0   1   0   1   1   1   0   0   0   0   1   0   1

>> [msg, error] = bchdeco (code_error, k, t)
msg =
0   1   0   0   0   1   1

error =  2
```

Ve zprávě jsme udělali tři chyby, a to na pozici 1., 2. a 13. Vidíme, že dekódována zpráva se nerovná s vyslanou. Ověřili jsme si, že kód dokáže opravit maximálně dvě chyby vzniklé při přenosu. Opět vidíme, že kód nám vypsal, že vznikly dvě chyby. Výsledná zpráva se nerovná s odeslanou.
Na těchto pár příkladech jsme si v krátkosti ukázali, jak BCH kódy fungují. Pro lepší pochopení jsme použili jednoduché zprávy a krátké kódy.

7 Vytvoření simulačního programu v GNU Octave

Pro zjištění funkčnosti a výše uvedené teorie byla vytvořena aplikace, která dokáže simulovat BCH kódy.
Program Octave poskytuje základní grafické rozhraní, které nám bohatě stačí pro simulaci těchto kódů. Simulace je zaměřená pouze na binární kódy. Při průběhu programu můžeme zadat zprávu, kterou chceme odeslat, ovlivnit počet chyb vzniklých ve zprávě, zobrazit si úspěšnost dekódování, ale také například to, jestli chceme v programu pokračovat nebo ne.

7.1 Zobrazení

Při spuštění skriptu s názvem *bakalářská práce*, si při přepnutí v dolní části obrazovky klikneme na *"command window"* , kde se nám zobrazí náš spuštěný skript.
Po přepnutí se nám zobrazí takováto stránka, kde jako první odrážku vidíme název našeho skriptu, popis programu. V našem případě tedy *"Test korekčních kódů - binární BCH"* a následně nás program vybízí k zadání zprávy.

```
>> bakalarska_prace
-------------------------------------------------------------------
Test korekcnich kodu - BCH
-------------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym binarnim cislem:
```

7.2 Zadání zprávy

Zpráva musí být zadaná v binárním tvaru, tedy kombinací jedniček a nul, mezi kterýma musí být mezera. Následná kombinace jedniček a nul musí být vložena do hranatých závorek z důvodu zadání zprávy v programu Octave. Vysílané zprávy se v prostředí Octave zadávají jako *"message = []"*. Zadáme zprávu délky od čtyř bitů až do délky 502. Důvod tady toho omezení je ten, že prvky do programu bereme z tabulky *"bchpoly"*, kde je nejmenší délka BCH(7,4) a nejdelší BCH(511,502).
Zadáme pro zjednodušení jednoduchou kombinaci binárního kódu ve tvaru BCH(7,4) ve tvaru *"[0 0 0 1]"* a potvrdíme tlačítkem enter.

```
>> bakalarska_prace
-------------------------------------------------------------------
Test korekcnich kodu - BCH
-------------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
```

```
za kazdym binarnim cislem: [0 0 0 1]
msg =

0   0   0   1

- 1 delka zpravy: 7, pocet chyb: 1
- Vyberte delku zpravy 1,2,3...:
```

7.3 Výběr délky

Vidíme, že naše zpráva má na výběr v *"bchpoly"* jen jednu kombinaci a to se zprávou délky sedm, která dokáže opravit jen jednu chybu. Tento krok jsme do programu přidali z důvodu, kdy by zpráva mohla být zakódována do různých délek kódů. Tento případ si ukážeme později.

```
>> bakalarska_prace
-----------------------------------------------------------------
Test korekcnich kodu - BCH
-----------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym binarnim cislem: [0 0 0 1]
msg =

0   0   0   1

- 1 delka zpravy: 7, pocet chyb: 1
- Vyberte delku zpravy 1,2,3...:
```

Po naší volbě jedna se nám zobrazil tvar zakódované zprávy. Vidíme, že první tři pozice jsou pro redundantní bity a následné čtyři pozice obsahuje námi zakódovanou zprávu.

7.4 Počet generujících se chyb

Program nám vypíše, kolik chyb dokáže tento kód opravit a kolik chceme chyb generovat. Generující chyby se nám v programu tvoří zcela náhodně. Zprávu a následné generování chyb do ní vložené jsme v programu zadali s opakováním sto. Jestli v našem programu zadáme počet chyb $chyb \leq 1$, program by nám měl dekódovat zprávu se 100 procentní úspěšností. Avšak zadáme-li počet chyb $chyb > 1$, kód by neměl opravit žádnou ze zpráv, a tedy schopnost oprav kódu by měla být 0 procent. Existují však kódové kombinace například BCH(15,5), které dokážou opravit i více chyb do zprávy vložené. Tyto kódy si nasimulujeme později.

```
>> bakalarska_prace
-------------------------------------------------------------------
Test korekcnich kodu - BCH
-------------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym binarnim cislem: [0 0 0 1]
msg =
0   0   0   1

- 1 delka zpravy: 7, pocet chyb: 1
- Vyberte delku zpravy 1,2,3...: 1
- Zakodovana zprava:   1   0   1   0   0   0   1
-------------------------------------------------------------------
- Kod dokaze opravit 1 chyb(y). Kolik chyb chcete generovat? 1
-------------------------------------------------------------------
- Procentualni uspesnost dekodovani 100 %
- Chcete pokracovat odznova? (0/1)
```

Po zadání jedné chyby vidíme, že kód opravil chybu se 100 procentní úspěšností.

7.5 Ověření opravné schopnosti kódu

Můžeme si ověřit, jestli tedy kód má schopnost opravit více, než jednu chybu.

```
>> bakalarska_prace
-------------------------------------------------------------------
Test korekcnich kodu - BCH
-------------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym binarnim cislem: [0 0 0 1]
msg =
0   0   0   1

- 1 delka zpravy: 7, pocet chyb: 1
- Vyberte delku zpravy 1,2,3...: 1
- Zakodovana zprava:   1   0   1   0   0   0   1
-------------------------------------------------------------------
- Kod dokaze opravit 1 chyb(y). Kolik chyb chcete generovat? 3
- Zadali jste pocet chyb vetsi, nez je kod schopen opravit.
- Chcete vypsat varianty, ktere se i pres to dokazi opravit? (0/1)1
```

```
-----------------------------------------------------------------
- Procentualni uspesnost dekodovani 0 %
- Chcete pokracovat odznova? (0/1)
```

Po zadání $t < chyb$ vidíme, že nás program upozorňuje, že jsme zadali větší počet chyb, než je ve schopnosti kódu opravit a následně se nás opět zeptá, jestli chceme vypsat varianty, které kód dokáže i přesto opravit. U této kódové kombinace se nám tento krok může zdát zbytečný, avšak u jiných kódových kombinacích, například již zmíněného kódu BCH(15,5) tento krok má svůj smysl. V tento moment samozřejmě nemá co vypsat, protože dekódování proběhlo s úspěšností 0 procent.
V posledním kroku se nás program zeptá, jestli chceme v simulaci pokračovat.

7.6 Výběr kódů

Při simulaci některých kódů, v našem případě v délce zprávy sedm, může nastat situace, která vyžaduje rozhodování jakou délku kódu vybrat, proto tento program zahrnuje tento druhý krok, díky kterému můžeme po vypsání délek rozhodnout, jakou kódovou kombinaci chceme použít.

```
>> bakalarska_prace
-----------------------------------------------------------------
Test korekcnich kodu - BCH
-----------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym binarnim cislem: [0 0 1 0 1 1 0]
msg =
0   0   1   0   1   1   0

- 1 delka zpravy: 15, pocet chyb: 2
- 2 delka zpravy: 63, pocet chyb: 15
- Vyberte delku zpravy 1,2,3...:
```

7.7 Simulace s opravením zadaných $chyb > t$

Některé kódy dokážou opravit i více chyb, než mají uvedené ve vztahu v tabulce *"bchpoly"*, Například již zmíněné BCH(15,5) kódy. Provedeme si tedy simulaci těchto kódů.

```
>> bakalarska_prace
-----------------------------------------------------------------
Test korekcnich kodu - BCH
```

```
-----------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym binarnim cislem: [0 0 1 0 1]
msg =
0   0   1   0   1

- 1 delka zpravy: 15, pocet chyb: 3
- Vyberte delku zpravy 1,2,3...: 1
- Zakodovana zprava:   0  0  0  0  1  1  1  0  1  1  0  0  1  0  1
-----------------------------------------------------------------
- Kod dokaze opravit 3 chyb(y). Kolik chyb chcete generovat? 4
- Zadali jste pocet chyb vetsi, nez je kod schopen opravit.
- Chcete vypsat varianty, ktere se i pres to dokazi opravit? (0/1)1
-----------------------------------------------------------------
- 1. Opraveny kod -   1  1  1  0  1  1  0  0  1  1  0  0  1  0  1
-----------------------------------------------------------------
- 2. Opraveny kod -   1  0  1  0  1  0  1  1  1  1  0  0  1  0  1
-----------------------------------------------------------------
- 3. Opraveny kod -   0  1  0  0  1  0  1  1  1  0  0  0  1  0  1
-----------------------------------------------------------------
- 4. Opraveny kod -   0  0  0  0  0  1  0  1  1  0  0  0  1  0  1
-----------------------------------------------------------------
- 5. Opraveny kod -   0  1  1  0  0  1  1  0  1  0  0  0  1  0  1
-----------------------------------------------------------------
- 6. Opraveny kod -   0  0  1  0  0  0  0  0  1  1  0  0  1  0  1
-----------------------------------------------------------------
- 7. Opraveny kod -   1  0  0  1  1  0  1  0  0  1  0  0  1  0  1
-----------------------------------------------------------------
- 8. Opraveny kod -   0  0  1  0  1  1  0  1  0  1  0  0  1  0  1
-----------------------------------------------------------------
- 9. Opraveny kod -   0  0  1  1  1  1  1  0  0  0  0  0  1  0  1
-----------------------------------------------------------------
- 10. Opraveny kod -  0  0  1  0  0  1  1  1  0  1  0  0  1  0  1
-----------------------------------------------------------------
- 11. Opraveny kod -  1  0  0  0  0  1  0  0  0  1  0  0  1  0  1
-----------------------------------------------------------------
- 12. Opraveny kod -  0  0  0  1  0  1  1  0  0  0  0  0  1  0  1
-----------------------------------------------------------------
- Procentualni uspesnost dekodovani 12 %
```

```
- Chcete pokracovat odznova? (0/1)
```

Můžeme vidět, že jsme zadali počet chyb = 4, i když kód dokáže opravit jen tři chyby, tak vidíme, že kód má 7 procentní úspěšnost i když jsme přesáhli schopnost oprav kódu. Vidíme, že jestli se chyba nevyskytuje ve zprávě kódu, dokáže zprávu zpět dekódovat. U všech sedmi případů ze sta tomu tak bylo, proto je úspěšnost 7 procent. Samozřejmě je úspěšnost ovlivněna i počtem vygenerovaných chyb ve zprávě. Kdyby se v desíti zprávách ze sta nevyskytovala chyba ve zprávě, kód by měl úspěšnost 10 procent.
Kód dokáže opravit i více vzniklých chyb ve zprávě, než-li 4. Ověřme si to.

```
>> bakalarska_prace
-----------------------------------------------------------------
Test korekcnich kodu - BCH
-----------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym
binarnim cislem: [0 0 1 0 1]
msg =
0   0   1   0   1

- 1 delka zpravy: 15, pocet chyb: 3
- Vyberte delku zpravy 1,2,3...: 1
- Zakodovana zprava:   0  0  0  0  1  1  1  0  1  1  0  0  1  0  1
-----------------------------------------------------------------
- Kod dokaze opravit 3 chyb(y). Kolik chyb chcete generovat? 7
- Zadali jste pocet chyb vetsi, nez je kod schopen opravit.
- Chcete vypsat varianty, ktere se i pres to dokazi opravit? (0/1)1
-----------------------------------------------------------------
- 1. Opraveny kod -   1  1  1  1  0  1  0  0  0  1  0  0  1  0  1
-----------------------------------------------------------------
- Procentualni uspesnost dekodovani 1 %
```

Na simulaci vidíme, že i při počtu chyb sedm, dokázal opravit jednu zprávu ze sta, kde se ve zprávě nevyskytovala chyba.

7.7.1 Další kódová kombinace

```
>> bakalarska_prace
-----------------------------------------------------------------
Test korekcnich kodu - BCH
```

```
-----------------------------------------------------------------
- Zadejte zpravu v binarni podobe do hranatych zavorek a s mezerou
za kazdym
binarnim cislem: [0 1 1 0 1 1]
msg =
0   1   1   0   1   1

- 1 delka zpravy: 31, pocet chyb: 7
- Vyberte delku zpravy 1,2,3...: 1
- Zakodovana zprava:   1  0  1  0  1  0  0  0  0  1  0  0  1  0  1
1  0  0  1  1  1  1  1  0  0  0  1  1  0  1  1
-----------------------------------------------------------------
- Kod dokaze opravit 7 chyb(y). Kolik chyb chcete generovat? 8
- Zadali jste pocet chyb vetsi, nez je kod schopen opravit.
- Chcete vypsat varianty, ktere se i pres to dokazi opravit? (0/1)1
-----------------------------------------------------------------
- 1. Opraveny kod -   1  0  1  0  0  0  0  0  1  1  1  0  1  1  1  1
1  1  1  0  1  1  1  1  0  0  1  1  0  1  1
-----------------------------------------------------------------
- 2. Opraveny kod -   1  0  1  0  0  1  0  0  0  0  0  1  1  0  0  1
0  1  0  1  1  0  1  0  0  0  1  1  0  1  1
-----------------------------------------------------------------
- 3. Opraveny kod -   0  0  0  0  0  0  0  0  1  1  0  0  1  0  1  1
0  1  1  0  0  0  1  0  0  0  1  1  0  1  1
-----------------------------------------------------------------
- 4. Opraveny kod -   0  0  0  0  1  0  1  0  0  1  0  0  1  0  0  1
0  1  0  1  1  1  1  1  1  0  1  1  0  1  1
-----------------------------------------------------------------
- 5. Opraveny kod -   1  1  1  0  1  0  0  0  0  0  0  1  1  1  1  1
0  0  0  0  1  1  0  0  1  0  1  1  0  1  1
-----------------------------------------------------------------
- 6. Opraveny kod -   0  0  1  0  1  0  0  1  0  0  1  0  1  0  0  1
1  0  1  1  1  1  0  0  1  0  1  1  0  1  1
-----------------------------------------------------------------
- 7. Opraveny kod -   1  1  1  0  0  0  1  0  0  1  1  0  0  1  1  1
0  0  0  0  1  1  1  0  0  0  1  1  0  1  1
-----------------------------------------------------------------
- 8. Opraveny kod -   1  0  0  0  0  0  0  0  0  0  1  1  0  0  1  1
1  0  1  0  1  1  1  0  0  0  1  1  0  1  1
```

```
-------------------------------------------------------------------
- 9. Opraveny kod -   1  1  1  1  0  0  0  0  1  1  0  0  1  0  1  0
0  1  1  1  0  1  1  0  1  0  1  1  0  1  1
-------------------------------------------------------------------
- 10. Opraveny kod -   0  0  1  0  0  0  0  1  0  1  1  0  1  0  1  0
0  0  1  1  1  0  1  1  1  0  1  1  0  1  1
-------------------------------------------------------------------
- 11. Opraveny kod -   1  0  1  0  0  1  0  0  1  1  0  1  1  0  1  1
0  0  0  0  1  1  0  1  0  0  1  1  0  1  1
-------------------------------------------------------------------
- 12. Opraveny kod -   1  0  1  0  1  0  0  0  1  0  0  0  1  0  0  0
0  1  1  0  0  1  1  1  0  0  1  1  0  1  1
-------------------------------------------------------------------
- 13. Opraveny kod -   1  0  1  0  0  0  0  0  0  1  1  1  1  0  1  0
0  1  0  1  0  1  0  0  0  0  1  1  0  1  1
-------------------------------------------------------------------
- Procentualni uspesnost dekodovani 13 %
- Chcete pokracovat odznova? (0/1)
```

Můžeme si zkusit všemožné kombinace a nasimulovat si je. V tomto případě vidíme BCH(31,6) kód, který dokázal opravit 8 chyb s 13 % úspěšností. Opět vidíme, že se chyba nevyskytovala v žádné ze zpráv. Kódových kombinací může být tolik, kolik obsahuje tabulka *"bchpoly"*.

8 Závěr

Jelikož mě předmět přenos dat velice zaujal, rozhodl jsem se, vzít si téma simulace BCH kódů pomocí open source nástroje Octave jako bakalářkou práci. Po sblížení se s tématem můžu říct, že to byla správná volba, jelikož mě téma opravdu zaujalo. Už dříve jsem se o přenos dat zajímal a díky bakalářské práci jsem se o samotných korekčních kódech používaných pro přenos dat dozvěděl ještě víc.

Cílem práce byla simulace BCH kódů v GNU Octave. Simulaci jsem provedl nejprve pomocí příkazového řádku a následně vytvořil program s grafickým výstupem, který slouží pro zjednodušení simulace BCH kódů, ale také pro lepší pochopení problematiky kódů.

Při studování BCH kódů jsem prošel obecnou teorii, vznik kódů, kódování a dekódování. U dekódování jsem zjistil, že existuje spousta dekódovacích algoritmů. Záměrem byl podrobnější popis jednoho z dekódovacích algoritmů. Po konzultaci s vedoucím práce Ing. Pavlem Nevludem jsem se rozhodl pro Peterson-Gorenstein-Zierlerův dekódovací algoritmus, který byl uveden jako první a všechny ostatní dekódovací algoritmy z něj vycházejí. Dekódování bylo nutné pro svou obtížnost konzultovat s lidmi z navazujících magisterských ročníků zaměřených na matematiku.

Při simulaci jsem zjistil vlastnost kódů, která nebyla popsána v žádné z uvedených literatur. Tato vlastnost dává kódu možnost opravit i více chyb, než je uvedeno v tabulce *"bchpoly"*. Proto jsem ve vytvořeném programu použil možnost při zadání většího počtu chyb, než dokáže program opravit podle výpisu z tabulky *"bchpoly"*, program umožňuje vypsat jednotlivé kódy s jejich procentuální úspěšností kódu. Tímto jsem zjistil, že jestliže se chyba nenaskytne přímo ve zprávě, ale jen v redundatních bitech, dokáže se přijatá zpráva opravit i když by podle tabulky *"bchpoly"* neměla.

Při psaní bakalářské práce jsem se naučil používat program Octave s doinstalování balíčku potřebných pro práci s prostředím. Program nadále pomohl k mému lepšímu porozumění BCH kódům a jejich funkcím.

V práci bych dále navázal na RS kódy, které jsou podskupinou BCH kódů a detailně vypracoval další dekódovací algoritmy, které jsou v práci jen zmíněny. Nadále by bylo možné zjistit zatížení procesoru při dekódování a v následných grafech ukázat rozdíly a výsledné hodnoty porovnat.

Petr Műller

Literatura

[1] W.CARY HUFFMAN, VERA PLESS, *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2010, ISBN 978–0521131704

[2] SCHMIDT HANSEN, Jesper. *GNU Octave: beginner's guide*. Birmingham: Packt Publishing, 2011. ISBN 978-1-849513-32-6.

[3] *Libor BARTO, Jiří TŮMA: Konečná tělesa* [online].[cit. 2016-22-03]. Dostupné z WWW: http://msekce.karlin.mff.cuni.cz/b̃arto/student/SkriptaKonTel.pdf

[4] *Petr ČÍKA: Obecný popis BCH kódu* [online].[cit. 2016-22-03]. Dostupné z WWW: http://www.elektrorevue.cz/clanky/06015/index.html

[5] *Doc.Ing. Karel NĚMEC, CSc.: Přehled nezbytných poznatků z lineární algebry* [online].[cit. 2016-22-03]. Dostupné z WWW: http://www.elektrorevue.cz/clanky/05015/index.html

[6] *Marie DEMLOVÁ: Galoisova tělesa* [online]. 2009 [cit. 2016-22-03]. Dostupné z WWW: http://math.feld.cvut.cz/demlova/teaching/avt/pred-a909.pdf

[7] *Pavel NEVLUD, Marek DVORSKÝ: Přenos dat* [online]. 2012 [cit. 2016-22-03]. Dostupné z WWW: http://www.person.vsb.cz/archivcd/FEI/PD/Prenos%20dat.pdf

[8] *Jiří TŮMA: Samoopravné kódy* [online].[cit. 2016-22-03]. Dostupné z WWW: http://msekce.karlin.mff.cuni.cz/ tuma/2002/NLinalg8.pdf

[9] J.KOTON,P. ČÍKA, V. KŘIVÁNEK: *Samoopravné Reed-Solomonovy kódy* [online]. 2006 [cit. 2016-22-03]. Dostupné z WWW: http://access.feld.cvut.cz/view.php?cisloclanku=200608000

[10] *Antonín JANČAŘÍK:Hammingovy kódy* [online].[cit. 2016-22-03]. Dostupné z WWW: http://class.pedf.cuni.cz/jancarik/download/hamming.pdf

[11] YUNGHSIANG S. HAN: *BCH Codes* [online].[cit. 2016-22-03]. Dostupné z WWW: http://web.ntpu.edu.tw/ yshan/BCH_code.pdf

[12] CLARKE *C. Reed-Solomon error correction*. British Broadcasting Corporation.2002

[13] R. E. BLAHUT, *Theory and Practice of Error Control Codes*. Reading, MA: Addison-Wesley, 1983, ISBN 978-0201101027

[14] *GNU Octave - Support Options* [online].[cit. 2016-22-03]. Dostupné z WWW: https://www.gnu.org/software/octave/support.html

[15] *Octave Forge - Extra packages for GNU Octave* [online].[cit. 2016-22-03]. Dostupné z WWW: http://octave.sourceforge.net/

A CD/DVD

DVD bude obsahovat zdrojový kód vytvořený v GNU Octave pro simulaci BCH kódů.